Office de la propriété intellectuelle du Canada

Un organisme d'Industrie Canada

Canadian Intellectual Property Office

An Agency of Industry Canada

PCT/CA 2005/000194

15 MARCH 2005 15·03.05

Bureau canadien des brevets

Certification

La présente atteste que les documents ci-joints, dont la liste figure ci-dessous, sont des copies authentiques des documents déposés au Bureau des brevets.

Canadian Patent Office

Certification

This is to certify that the documents attached hereto and identified below are true copies of the documents on file in the Patent Office.

Specification, as originally filed, with Application for Patent Serial No: **2,471,929**, on June 22, 2004, by **CHRIS M. DAVIES,** for "System and Method for a Self-Organizing, Reliable, Scalable Network".

[illegible]

Agent certificateur/Certifying Officer

March 15, 2005

Date

Canada

OPIC CIPO

System and Method for a Self-Organizing, Reliable, Scalable Network

This network system enables individual nodes in the network to coordinate their activities such that the sum of their activities allows communication between nodes in the network.

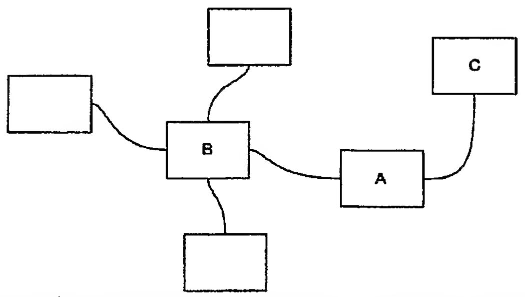
This network is similar to existing ad-hoc networks used in a wireless environment. The principle limitation of those networks is the ability to scale past a few hundred nodes. This method overcomes that scaling problem.

Note to Readers: examples are given throughout this document in order to clarify understanding. These examples, when making specific reference to numbers, other parties' software or other specifics, are not meant to limit the generality of the method and system described herein

Nodes

Each node in this network is directly connected to one or more other nodes. A node could be a computer, network adapter, switch, or any device that contains memory and an ability to process data. Each node has no knowledge of other nodes except those nodes to which it is directly connected. A connection between two nodes could be several different connections that are 'bonded' together. The connection could be physical (wires, etc), actual physical items (such as boxes, widgets, liquids, etc), computer buses, radio, microwave, light, quantum interactions, etc.

No limitation on the form of connection is implied by the inventors.



Node A is directly connected to nodes B and C
Node C is only connected to Node A
Node B is directly connected to four nodes

'target nodes' are a subset of all directly connected nodes. Only 'target nodes' will ever be considered as possible routes for messages (discussed later).

Nodes and Messages

A node must have a globally unique identifier (GUID). This GUID is used to identify the node on the network and allow other nodes to route packets to this node. If a node does not need to have packets routed to it, it does not need a GUID.

A node may keep this GUID permanently, or may generate a new GUID on startup. Node A can only send a message to node B if Node A knows the GUID of node B. A node may have multiple GUID's in order to emulate several nodes with different services running on each node.

Messages are sent to a certain GUID (that represents a machine), and to a particular port number on that machine (similar to TCP/IP). We refer to the machine where messages are sent to as the destination node. This is to separate it from the 'target node' that is the next best step towards the destination node.

If a node knows about a destination node it will tell those nodes it is connected to about that node. (discussed in detail later). The only node that knows the final destination for a message is the node that is final destination for that message. A node never knows if the node it passes a message to the ultimate destination for that message.

A destination node is the ultimate destination for a message.

At no point does any node attempt to build a global network map, or have any knowledge of the network as a whole except of the nodes it is directly connected to. The only knowledge it has is that another node exists, and which directly connected node is its next best step towards that node.

When a node first connects to another node in the network it tells the other node two things:

1. A Tie Breaker Number
   This is a very large random number used as a tie-breaker, in case there are two equal node choices, or both nodes choose each other as 'target nodes' for the same destination node at the same time.
2. Destination Node Count
   This tells the node that this node is connecting to the maximum number of other nodes that this node wants to know about. This is used to ensure that this node is not overrun with node updates.

Event though this message is sent right at the start, it can be sent later as well in order to reduce or increase the destination node counts. This message will be sent immediately with a new tie breaker number, if the directly connected node sends an identical tie-breaker number as another directly connected node.

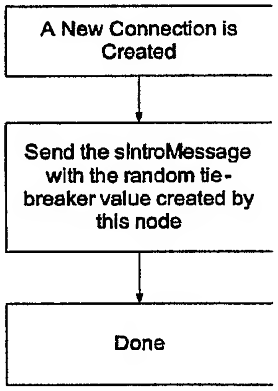
It is important that each node tells the same tie-breaker number to every directly connected node.

If a node needs to adjust the 'destination node count' number it should send the same 'tie-breaker' number.

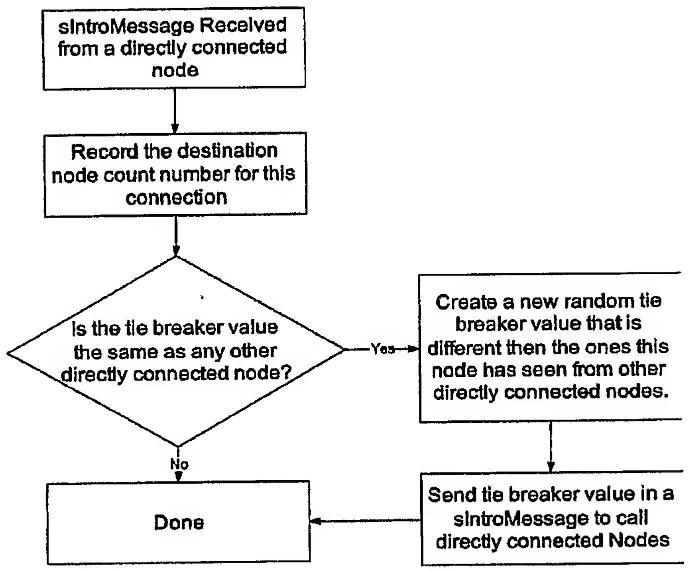
The structure used looks like this:

```
struct sIntroMessage {

        // the number used to break ties
        int             uiTieBreakerNumber;

        // the number of destination nodes this node wants to know about.
        int             uiDestinationNodeCount;
}
```

Below is a flowchart of the initialization process after a connection has been established:


A New Connection is Created
Send the sIntroMessage with the random tie-breaker value created by this node
Done

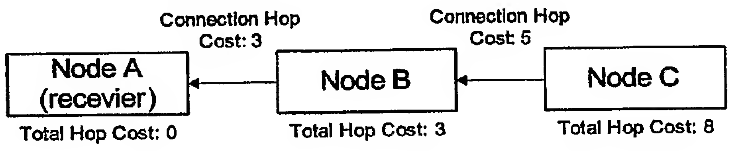
Below is a flowchart of what happens when an sIntroMessage is received from the directly connected node.


sIntroMessage Received from a directly connected node
Record the destination node count number for this connection
Is the tie breaker value the same as any other directly connected node?
Yes
Create a new random tie breaker value that is different then the ones this node has seen from other directly connected nodes.
No
Send tie breaker value in a sIntroMessage to call directly connected Nodes
Done

Calculation Of Hop Cost

'Hop Cost' is an arbitrary value that allows the comparison of two or more routes. In this document the lower the 'hop cost' the better the route. This is a standard approach, and someone skilled in the art will be aware of possible variations.

Hop Cost is a value that is used to describe the capacity or speed of the connection. It is an arbitrary value, and should not change in response to load. Hop Cost is additive along a connection.



In this example, node C has a total hop cost of 8 since connections between node A and B and node B and C total 8.

A lower hop cost should represent a higher capacity connection, or faster connection. These hop costs will be used to find a path through the network using a Dykstra like algorithm.

End User Software

This approach has been designed to act in a similar manner to standard TCP/IP connections. If a node wishes to connect to another node it will initiate a connection to that nodes name (in this case a GUID, in the case of TCP/IP an IP address), and a port number.

In TCP/IP when a node is turned on, it does not announce its presence to the network. It does not need to because the name of the node (IP address) determines its location. In this approach, the node needs the network to know that it exists, and provide the network with a guaranteed path to itself. This is discussed in much greater detail later.

When end user software (EUS) wishes to establish a connection, it does so in a manner very similar to TCP/IP. In TCP/IP the connection code looks similar to this:

```
SOCKET  sNewSocket = Connect(IP Address, port);
```

With this approach, the 'IP Address' is replaced with a GUID.

```
SOCKET sNewSocket = Connect(GUID,port).
```

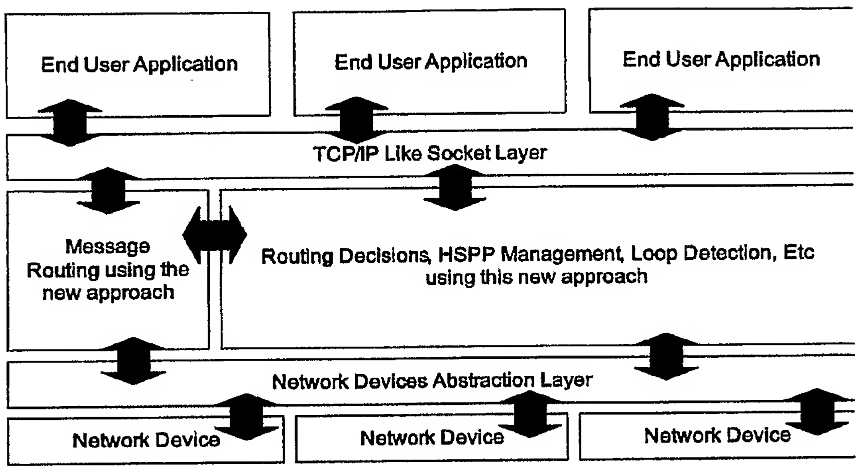
In fact, if the IP Address can be guaranteed to be unique, then the IP address could serve as the GUID, providing a seamless replacement of an existing TCP/IP network stack with this new approach.

Once the connection has been requested, the network will determine a route to the destination (if such a route exists), and continually improve the route until an optimal route has been found.

The receiving end will look identical to TCP/IP, except a request to determine the IP address of the connecting node will yield a GUID instead.

This approach provides the routing through the network, someone skilled in the art could see how different flow control approaches might work better in networks with lots of change versus networks with not very much change.

Below is a diagram indicating where this routing method fits:


End User Application
End User Application
End User Application
TCP/IP Like Socket Layer
Message Routing using the new approach
Routing Decisions, HSPP Management, Loop Detection, Etc using this new approach
Network Devices Abstraction Layer
Network Device
Network Device
Network Device

Someone skilled in the art should see that this new routing approach allows a TCP/IP like interface for end user applications. This is and example not meant to limit this routing approach to any particular interface or application.

Initial Destination Node Knowledge

When a node is connected to the network, the system needs a way to find a path through the network. This path is through a series of directly connected nodes. No node is aware of the complete path. Each node is only aware of its next best step to the ultimate destination.

When a node is first connected to the network, it tells all directly connected nodes:

1. The name of the Node
   This is a name that is unique to this node. Two nodes independently created should never have the same name.
2. Hop Cost
   This is a value that describes how good this route to the ultimate receiver. Each hop in the path is assigned a hop cost, the uiHopCost is a reflection of the summation of these individual hop costs between the node that provided the update and the ultimate receiver.
3. Distance from data flow
   Discussed Later. Very similar to 'Hop Cost', except that it describes how far this node is from a data flow. This can be used to decide which node updates are 'more important'. A node that is in the data flow has a 'uiHopCostFromFlow' of 0. The uiHopCostFromFlow is a summation of all the hop costs between the node that told this node of this destination node and a data flow for that destination node

4. Target Node
   We'll tell the node we're sending this update to if it is a 'target' node for messages sent to a destination node. This creates a 'poison reverse'. If two node
5. In Data Stream
   If this node is in the data stream, then we'll need to tell our directly connected node that is the 'target node' for this destination node that it is in the data stream. This is discussed in more detail later.

This update takes the structure of:

```
struct sDestNodeUpdate {

        // the name of the node. Can be replaced with a number
        // (discussed later)
        sNodeName               nnName;

        // the hop cost this node can provide to the ultimate
        // receiver
        unsigned int            uiHopCost;

        // calculated in a similar fashion to 'uiHopCost'.
        // and records the distance from the data flow for this
        // node.
        unsigned int            uiHopCostFromFlow;

        // if the node we're sending this to is a target node for this
        // destination node, then we'll set this to true;
        boolean                 bIsTargetNode;

        // if the we're in the data stream, and this directly connected
        // node is the target node for this destination node
        Boolean                 bIsInDataStream;
};
```

Regardless of whether this is a previously unknown destination node or an update to an already known node the same information is sent.

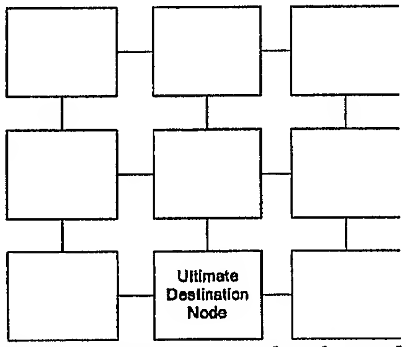
If this is the first time a directly connected node has heard about that destination node it will choose the node that first told it as its 'target node' for messages to that destination node. A node will only send EUS messages to a node or nodes that are 'target nodes', even if other nodes tell it that they too provide a route to the destination node.

In this fashion a network is created in which every node is aware of the destination node and has a non-looping route to the destination node through a series of directly connected nodes.

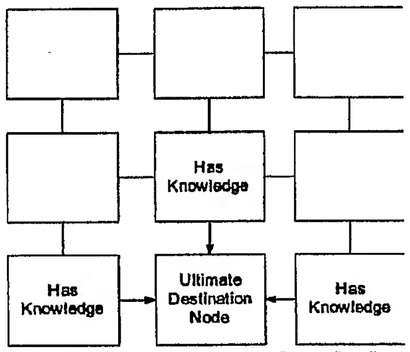
If a node A has chosen node B as a 'target node', node A will tell node B that it is a 'target node' for that particular destination node. This will create a 'poison reverse' that will stop many loops from being created.

If both nodes tell each other at the same time that they are target nodes, the node with the highest 'tie-breaker' number gets to keep their choice, and the other node must make another choice.

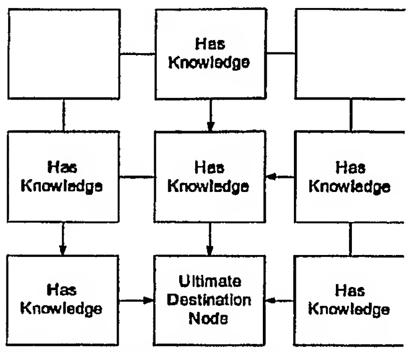
Below is a series of diagram illustrating the spread of destination node knowledge.



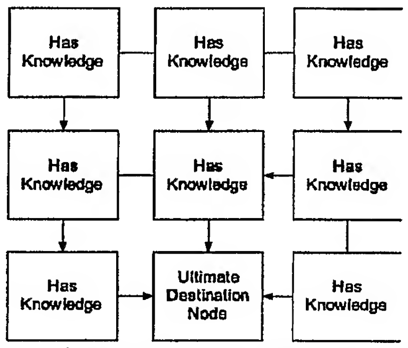
Step1: The ultimate destination connects to several nodes in the network and prepares to send an initial destination node update to its directly connected neighbor nodes.



Step2: As destination node knowledge spreads, each node that gets told of the destination node selects the node that told it as the 'target node' for messages routed to that destination node.



Step3: Knowledge of the destination node continues to spread, with each node that is told about the destination node choosing the node that told it. If another node provided it with a better hop cost later, the node would choose the directly connected node with the better hop cost.



Step4: In this step all nodes in the network are aware of the destination node and have a 'target node' that they can send messages that are destined for the destination node.

Note: the linkages between nodes and the number of nodes in this diagram are exemplar only, whereas in fact there could be indefinite variations of linkages within any network topography, both from any node, between any number of nodes.

At no point does any node in the network attempt to gather global knowledge of network topology or routes.

Even if a node has multiple possible paths for messages it will only send messages along the node that it has chosen as its 'target node'.

The value 'uiHopCostFromFlow' can be used to help determine which nodes updates are more important (See Propagation Priorities). If the node is 100 units from the flow for destination node A, and 1 unit away from the flow for destination node B, it will send a node update about destination node B, before an update about destination node A.

The uiHopCostFromFlow is also used to determine which destination node updates are to be sent, and which are not to be sent. During the initialization process, each node tells all its directly connected nodes how many destination nodes it wants to know about. Nodes that receive these updates must respect these limits.

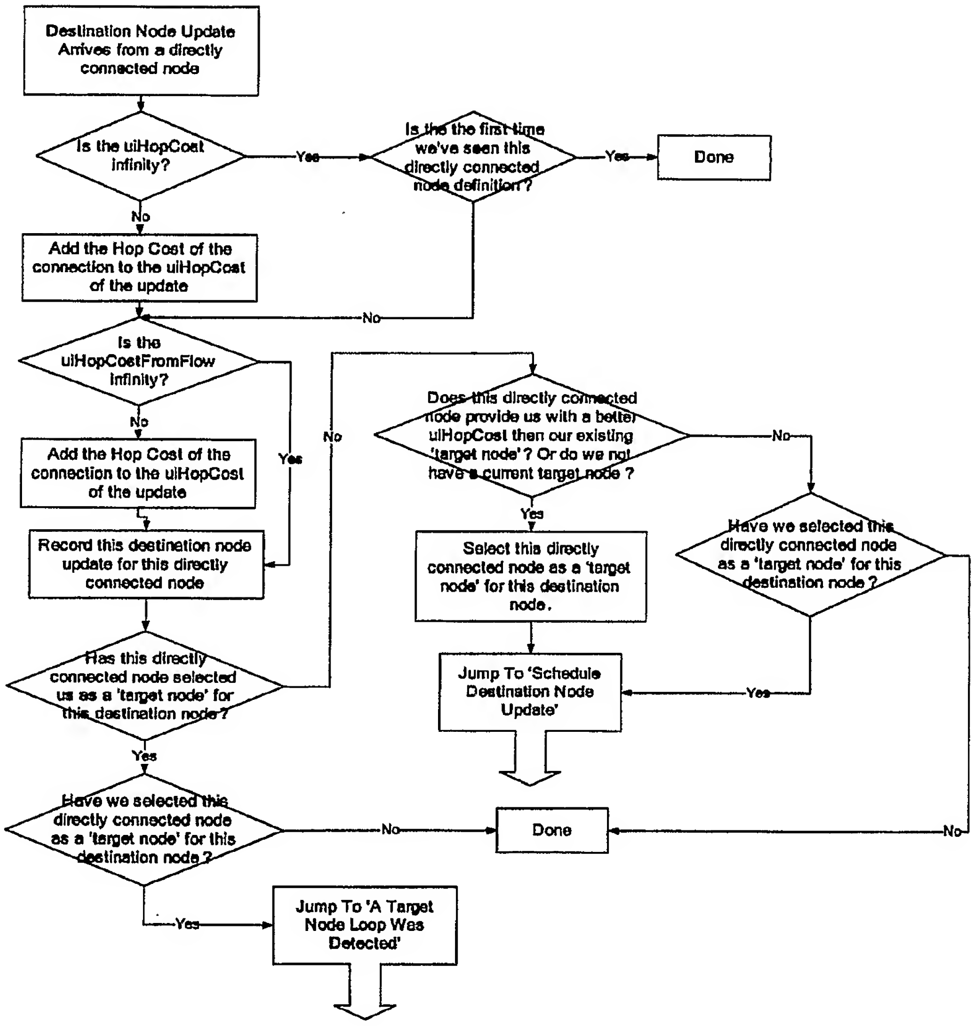
Destination nodes are ranked by uiHopCostFromFlow in order to determine which updates should be sent and which should not be.

If a destination node update that was sent has its uiHopCostFromFlow change so that it should no longer be sent, an update with a 'uiHopCost' of infinity is sent to the node. This will allow that node to reuse the memory associated with that destination node.

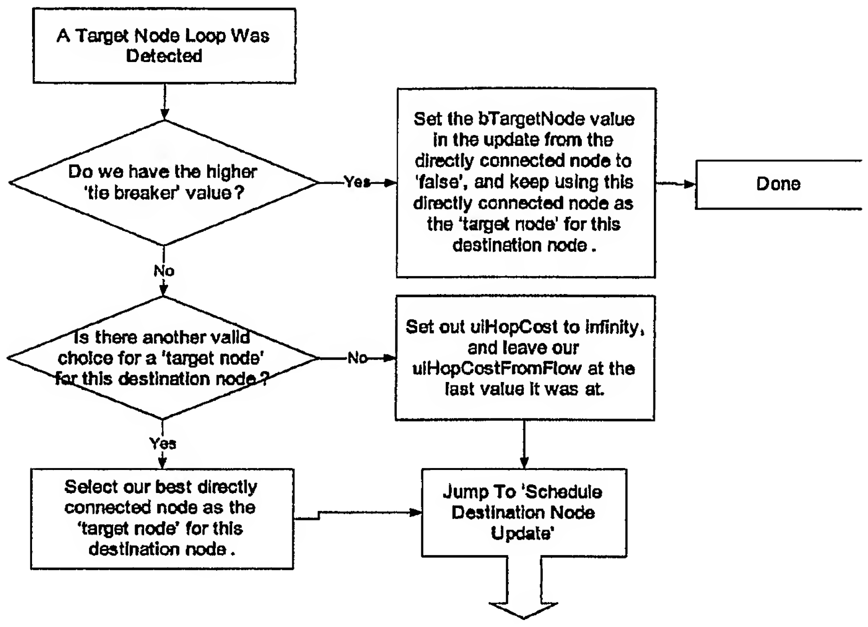
A node that is told about a new destination node with uiHopCost of infinity will ignore that destination node.

When an sDestNodeUpdate arrives from a directly connected node, the hop cost of that connection is added to both the uiHopCost and uiHopCostFrom flow before it is processed by the node. The exception is if the uiHopCost or uiHopCostFromFlow is infinity then nothing is added to those values.

This flowchart outlines the basic process of dealing with an update.


Destination Node Update Arrives from a directly connected node
Is the uiHopCost Infinity?
Yes
Is the the first time we've seen this directly connected node definition?
Yes
Done
No
Add the Hop Cost of the connection to the uiHopCost of the update
No
Is the uiHopCostFromFlow Infinity?
No
Add the Hop Cost of the connection to the uiHopCost of the update
Yes
Record this destination node update for this directly connected node
Has this directly connected node selected us as a 'target node' for this destination node?
No
Does this directly connected node provide us with a better uiHopCost then our existing 'target node'? Or do we not have a current target node ?
No
Yes
Select this directly connected node as a 'target node' for this destination node.
Have we selected this directly connected node as a 'target node' for this destination node ?
Jump To 'Schedule Destination Node Update'
Yes
Yes
Have we selected this directly connected node as a 'target node' for this destination node ?
No
Done
No
Yes
Jump To 'A Target Node Loop Was Detected'

This next flowchart outlines the process of dealing with a target node loop. This is where two directly connected nodes selected each other as 'target nodes' for a particular destination node.


A Target Node Loop Was Detected
Do we have the higher 'tie breaker' value?
Yes
Set the bTargetNode value in the update from the directly connected node to 'false', and keep using this directly connected node as the 'target node' for this destination node.
Done
No
Is there another valid choice for a 'target node' for this destination node?
No
Set out uiHopCost to infinity, and leave our uiHopCostFromFlow at the last value it was at.
Yes
Select our best directly connected node as the 'target node' for this destination node.
Jump To 'Schedule Destination Node Update'

The node maintains a list of all destination nodes that are known about in sorted order by the uiHopCostFromFlow of each destination node. In order to preserve a specific order, and a unique value for lookups, the list is keyed to a double value, where the number to the left of the decimal place is the uiHopCostFromFlow and the number to the right of the decimal place is the internal destination node number.

For example, if the uiHopCostFromFlow was 129 and the internal node number was 9214 the number used in the ordered list would be:

129.9214

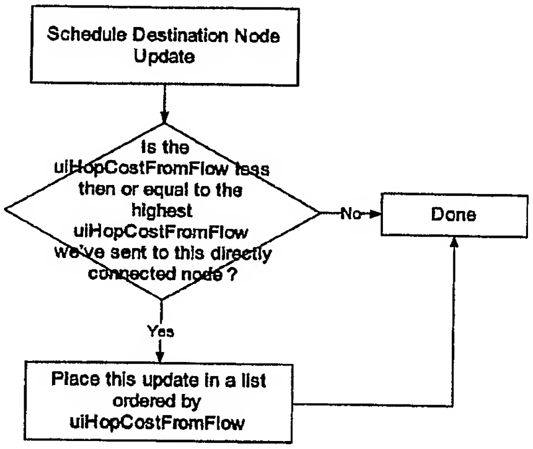
This special number is called the 'Sorted Order Key'.

The node also maintains a set of structures on each directly connected node. Part of the structure might look like this:

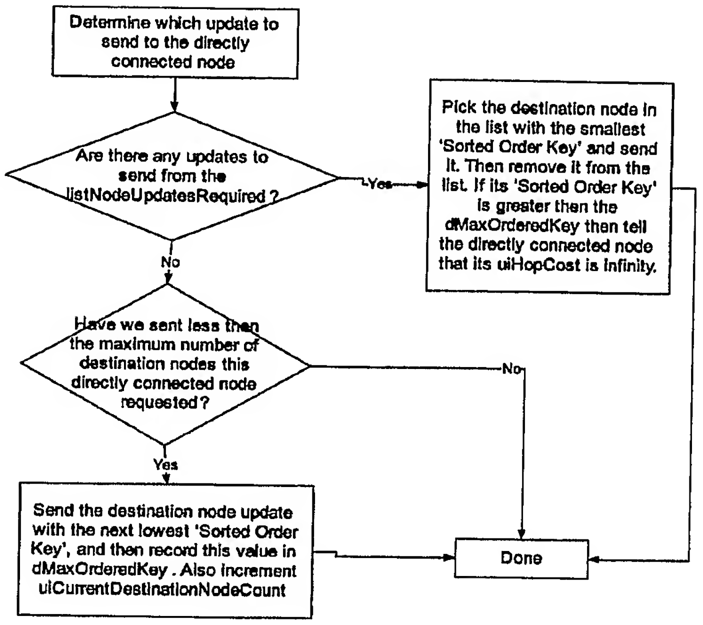
```
struct  sDirectlyConnectedNode {

        ....

        // the maximum number of destination nodes that the
        // directly connected node wishes to be told about
        int             nMaxDestinationNodeCount;
        // the current number of destination nodes we've told
        // the directly connected node about
        int             nCurrentDestinationNodeCount;

        // 'Sorted Order Key' of the highest uiHopCost node we've
        // sent to this directly connected node
        double          dMaxOrderedKey;

        // a list of destination nodes that are ordered by their
        // 'Sorted Order Key' that need to be sent to this directly
        // connected node
        SortedList      listNodeUpdatesRequired;

        ....
};
```

Structures similar to this are needed to ensure that when sending destination node updates to a directly connected node we're only sending those node updates that are most directly relevant to that node. By definition, those destination node update that are most relevant are those updates that have the smallest uiHopCostFromFlow values.

This next flowchart outlines how scheduling destination node updates works. This process is run on a per directly connected node basis.


Schedule Destination Node Update
Is the uiHopCostFromFlow less then or equal to the highest uiHopCostFromFlow we've sent to this directly connected node?
No
Done
Yes
Place this update in a list ordered by uiHopCostFromFlow

This next process is run when determining which destination node updates should be sent in the bandwidth throttled control packets.


Determine which update to send to the directly connected node
Are there any updates to send from the listNodeUpdatesRequired ?
Yes
Pick the destination node in the list with the smallest 'Sorted Order Key' and send it. Then remove it from the list. If its 'Sorted Order Key' is greater then the dMaxOrderedKey then tell the directly connected node that its uiHopCost is Infinity.
No
Have we sent less then the maximum number of destination nodes this directly connected node requested?
No
Yes
Send the destination node update with the next lowest 'Sorted Order Key', and then record this value in dMaxOrderedKey . Also increment uiCurrentDestinationNodeCount
Done

If we've sent the maximum number of destination node updates to the directly connected node, when we get a destination node update that was not able to be sent before (because it had a 'Sorted Order Key' that was too high), but now has a 'sorted order key' that is below dMaxOrderedKey for this connection we'll place this destination node update and the node with the 'Sorted Order Key' of dMaxOrderedKey in the listNodeUpdatesRequired. Then we'll find the next lowest 'Sorted Order Key' in the node's global node list and set dMaxOrderedKey to that value.

This will cause the new destination node update to be sent out, and the highest 'Sorted Order Key' update that was sent out before, to be sent out again with a uiHopCost of infinity. This allows the directly connected node to forget about that destination node.

Nodes In The Data Stream

A node is considered in the data stream if it is on the path for data flowing between an ultimate sender and ultimate receiver. A node knows it is in the data stream because a directly connected node tells it that it is in the data stream. Node A may only tell node B that it is in the data stream for messages to node C if node A has also told node B that node B is the 'target node' for messages to node C. If node B has been told it is in the data stream for messages to node C, then it will tell its directly connected node (node D) that it has selected as the best route to node C, that node D is in the data stream.

The first node to tell another node that it is 'in the data stream' is the node where the EUS resides that is establishing a connection to that particular node. For example, if node A wants to establish a connection to node C, and node B is the target node that node A has selected as its best route to node C, node A would tell node B that it is in the data flow for node C.

If node A tells node B it is no longer in the data stream for node C, then node B will tell its directly connected node that was used as the next best step to node C (node D) that it is no longer in the data stream.

Basically, if a node is not in the data stream any more it tells the node it has chosen as a 'target node' that it is not in the data stream any more.

All nodes used in communication between nodes are marked as in the data stream.

When a node is in the data stream for a destination node, it will set its uiHopCostFromFlow for that destination node to 0. This allows updates for that destination node to spread quickly around the data path. This allows the data path to continually locate better routes.

A node will maintain a count of the number of directly connected nodes that have told it that they are in the data stream. If the count drops to zero, it will tell its 'target node' for that destination that it is no longer in the data stream.

Node Name Optimization and Messages

Every node update and messages needs to have a way to identify which destination node it references. Node names and GUIDSs can easily be long, and inefficient to send with every message and node update. Nodes can make these sends more efficient by using numbers to represent long names.

For example, if node A wants to tell node B about a destination node named 'THISISALONGNODENAME.GUID', it could first tell node B that:

1 = 'THISISALONGNODENAME.GUID'

A structure for this could look like:

```
struct sCreateQNameMapping {
        int             nNameSize;
        char            cNodeName[Size];
        int             nMappedNumber;
};
```

Then instead of sending the long node name each time it wants to send a destination node update, or message - it can send a number that represents that node name. When node A decides it no longer wants to tell node B about the destination node called 'THISISALONGNODENAME.GUID', it could tell B to forget about the mapping.

That structure would look like:

```
struct sRemoveQNameMapping {
        int             nNameSize;
        char            cNodeName[Size];
        int             nMappedNumber;
};
```

Each node would maintain its own internal mapping of what names mapped to which numbers. It would also keep a translation table so that it could convert a name from a directly connected node to its own naming scheme. For example, a node A might use:

1 = 'THISISALONGNODENAME.GUID'

And node B would use:

632 = 'THISISALONGNODENAME.GUID'

Thus node B, would have a mapping that would allow it to convert node A's numbering scheme to a numbering scheme that makes sense for node B. In this example it would be:

Node A	Node B
1	632
...	...
...	...

Using this numbering scheme also allows messages to be easily tagged as to which destination node they are destined for. For example, if the system had a message of 100 bytes, it would reserve the first four bytes to store the destination node name the message is being sent to, followed by the message. This would make the total message size 104 bytes. An example of this structure also includes the size of the message:

```
struct sMessage {
        int             uiNodeID;
        int             uiMsgSize;
        char            cMsg[uiMsgSize];
}
```

When this message is received by the destination node, that node would refer to its translation table to decide which directly connected node this message should be sent to.

These quick destination numbers could be placed in a TCP/IP header by someone skilled in the art.

When To Remove Mappings

Mapping the GUID to an internal number allows the node to process messages much more quickly, however in order for this to work well it must also tell the directly connected nodes about this number mapping.

If a node wants to reclaim memory used to store information about a particular destination node M that has gone to infinity, then it needs to tell those directly connected nodes to forget about its particular internal mapping. (see above), and remove any message destined for destination node M.

Once those directly connected nodes tell this node that it too can forget about their mappings from this destination node to their internal number then this node can remove all knowledge of that particular destination node.

Simpler Fast Routing

An optimization would be add another column to the mapping table indicating which directly connected node will be receiving the message:

Thus node B, would have a mapping that would allow it to convert node A's numbering scheme to a numbering scheme that makes sense for node B. In this example it would be:

Node A	Node B	Directly Connected Node Message is Being Sent To
1	632	7
...	...	
...	...	

This allows the entire routing process to be one array lookup. If node A sent a message to node B with a destination node 1, the routing process would look like this:

1. Node B create a pointer to the mapping in question:
   sMapping *pMap = &NodeMapping[pMessage->uiNodeID];
2. Node B will now convert the name:
   pMessage->uiNodeID = pMap->uiNodeBName;
3. And then route the message to the specified directly connected node:
   RouteMessage(pMessage,pMap->uiDirectlyConnectedNodeID;

For this scheme to work correctly, if a node decides to change which directly connected node it will route messages to a directly connected node, it will need to update these routing tables for all directly connected nodes.

Path to Destination Node Removed

If the connection between node A and directly connected node (node B) is broken (or the directly connected node is removed from the network), node A will need to find new 'target nodes' for those destination nodes that were using node B.

Node A will pick the lowest non-infinity directly connected node that has not picked node A as a 'target node'. If node A is unable to find a directly connected node that matches this criteria then it will set that destination nodes' uiHopCost to infinity. When it sets that destination nodes uiHopCost to infinity it will keep the last known uiHopCostFromFlow value.

Since we use a poison reverse, we'll eliminate most loops immediately. Those loops we don't eliminate will cycle upwards, which in existing approaches will eat up large amounts of network bandwidth and cause the network to stop working.

In our approach we use the value uiHopCostFromFlow to determine which updates are sent to which node, and in what order those updates are sent. If a loop is created, there are two possibilites:

Possibility One
The actual destination node has been removed from the network (or can't be connected to from here).

In this case, there is no lower latency that can be used to resolve this loop. However, since uiHopCostFromFlow will spiral upwards as quickly as uiHopCost, the loop will be very quickly have its update priority set as low as possible, and it will be removed from the list of destination nodes sent to low memory nodes.

In time, the system will detect that a loop is present an remove it. However, time is not pressing, since almost no resources are utilized to support the loop.

This is discussed further in 'Resolving Accidentally Created Loops'.

Possibility Two
The actual destination node is reachable from here, however lower uiHopCost updates from that destination have not reached the loop yet to resolve it.

Initially the loop will cycle up the same way as it does it possibility one, and very quickly render itself irrelevant. If the directly connected nodes are at all memory limited, it will be removed from the list of destination nodes sent to those directly connected nodes.

When the uiHopCost that comes from the actually destination node reaches the loop, the loop will be automatically resolved, since the uiHopCost will be less that the increasing spiral of uiHopCosts in the loop.

If the uiHopCost from the actual destination node reaches the loop after the loop has been detected and removed, it will simply act as if the update was spreading to the nodes in the ex-loop for the first time.

This is discussed further in 'Resolving Accidentally Created Loops'.

Converging on Optimal Paths

Once a node has 'target node' for a destination node, it will begin looking for a better route. A better route is a route with a lower hop cost.

A node will select as a 'target node' (for a particular destination node) any node that offers it a lower hop cost then its current 'target node' for a particular destination node. It will never pick a directly connected node that uses this node as its 'target node'. This is the poison reverse in action.

If it is unable to find a target node that is non-infinity, and doesn't trigger the poison reverse, then it will set its uiHopCost to infinity and keep the last used uiHopCostFromFlow.

This process is very similar to Dykstra's algorithm.

Pipe Capacity and Hop Cost

Pipes with low capacity should be assigned a high uiHopCost. Those with high capacity should be assigned a low hop cost.

It is important that uiHopCost of a pipe has an approximately direct relationship to its capacity. For example, a 1Mbit pipe would need to have 10 times the uiHopCost of 10Mbit pipe.

In order to dramatically reduce the odds of triggering a false loop removal run, a very small, random variation should be introduced into the hop cost assignment. This variation should be less then 1%.

Someone skilled in the art will be able to assign hop costs.

Resolving Accidentally Created Loops

If a loop is created, it can be detected because uiHopCost will cycle upwards continually with even steps. If the node is not in the data stream, uiHopCostFromFlow will also cycle upwards with the same even steps.

If the loop is not in the data stream, it is not critical that the loop gets solved quickly. Since the uiHopCostFromFlow will quickly increase, these loop updates will only be sent when no other 'real' updates of any sort need to be sent.

However, they do need to be resolved in order to conserve memory, and in the case of loops while in the data stream, to ensure proper connectivity.

Since this approach makes use of a poison reverse, most loops will never be created.

If a loop is detected when uiHopCost (and when not in the data stream) uiHopCostFromFlow increase by the same amount more then 10 times for example (or a number of times determined by someone skilled in the art for their application) in a row.

The time between these updates from the 'target node' for this particular destination node are also recorded. This value is called the 'Target Node Update Delay Time'

When this happens, the node will force its uiHopCost to infinity until it sees its directly connected target node for that destination node has a uiHopCost of infinity or a value less then it currently has, or an amount of time equal to 10 times the 'Target Node Update Delay Time' has elapsed.

At that point it will pick the directly connected node with the lowest non-infinity uiHopCost and no poison reverse as a 'target node' for that destination node. If no such node exists, it will maintain its uiHopCost at infinity.

Large Networks

In very large networks with a large variation in interconnect speed and node capability different technique need to be employed to ensure that any given node can connect to any destination node in the network, even if there are millions of nodes.

Using the original method, knowledge of a destination node will spread quickly through a network. The problem in very large networks that contain large numbers of destination nodes is three fold:

1. The bandwidth required to keep every node informed of all destination nodes grows to a point where there is no bandwidth left for data.

2. Bandwidth throttling on destination node updates used to ensure that data can flow will slow the propagation of destination node updates greatly.
3. Nodes with not enough memory to hold every destination node will potentially cut off possible ultimate receivers from large parts of the network.

The solution is found by determining what constitutes the 'core' of the network. The core of the network will most likely have nodes with more memory and bandwidth then an average node, and most likely to be centrally located topologically.

Since this new network system does not have any knowledge of network topology, or any other nodes in the network except the nodes directly connected to it, nodes can only approximate where the core of the network is.

This is done by examining which directly connected node is a 'target node' for the most destination nodes. A directly connected node is picked as a 'target node' because it has the lowest uiHopCost, and the lowest uiHopCost will generally be provided by the directly connected node that is closest to the ultimate destination node. If a node is used as a 'target node' for more destination nodes then any other directly connected node, then this node is probably a step toward the core of the network.

If there is a tie between a set of directly connected nodes for who was picked as the 'target node' for the most destination nodes, the directly connected node with the highest 'tie-breaker' value (which was passed during initialization) will be selected as the next best step towards the core. This mechanism will ensure that there are no loops (besides Node A to Node B to node A type loops).

Since nodes not at the core of the network will generally not have as much memory as nodes at the core, they may be forced to forget about an ultimate receiver that relies on them to allow others to connect. If they did forget, no other node in the network would be able to connect to that ultimate receiver.

In the same way, a node that is looking to establish a connection with an ultimate receiver faces the same problem. The destination node definition that it is looking for won't reach it fast enough - or maybe not at all if it is surrounded by low capacity nodes.

The solution to these problems is to set up a high speed propagation path (HSPP) between the node that is the receiver or sender to the core of the network. A HSPP is tied to a particular destination node name or class of destination node names. If a node is in a HSPP for a particular destination node it will immediately process and send:

1. Initial knowledge of the destination node
2. When the destination node uiHopCost goes to infinity

3. When the destination node uiHopCost moves from infinity to some other value

To those nodes directly in the HSPP. This will ensure that all nodes in the HSPP will always know about the destination node in question, if any one of those nodes can 'see' the destination node.

Node knowledge is not contained in the HSPP. The HSPP only sets up a path with a very high priority for knowledge of a particular destination node. That means, that any destination node update that is in one of the previous three categories will be immediately sent.

There are two types of HSPP's. One type pushes knowledge of a destination node towards the core, the second pulls knowledge of a destination towards the node that created the HSPP.

When an ultimate destination node is first connected to the network, it will create an HSPP based on its node name. This HSPP will be of the type that pushes knowledge of this node towards the core of the network. It will send this HSPP to the directly connected node that it has decided is the next best step towards the core. The HSPP created by the ultimate destination node will be maintained for the life of the node. If the node is disconnected, the HSPP will be removed.

If an ultimate sender is trying to connect to an ultimate destination node, and does not have knowledge of the destination node, it will create an HSPP. This HSPP is the type that will pull knowledge of the destination node back towards the node that wants to connect to the destination node. An HSPP of this type will only travel until it reaches knowledge of the destination node referred to in the HSPP.

As soon as a connection is established with the destination node, the ultimate sender of the node will remove the HSPP.

An HSPP is not a path itself, rather it forces nodes on the path to retain knowledge of the node in question.

<u>How an HSPP is Established and Maintained</u>

If a node is told of an HSPP it must remember that HSPP until it is told to forget that HSPP, or the connection between it and the node that told it of the HSPP is broken.

Each node must store as many HSPP's as are given to it.

In most systems the amount of memory available on nodes will be such that it can assumed that there is always enough memory, and that no matter how many HSPP's pass through a node it will be able to store them all. This is even more likely because the number of HSPP's on a node will be roughly related to how close this node is to the core, and a node is usually not close to a core unless it has lots of capacity, and therefore probably lots of memory.

UR = Ultimate Receiver
US = Ultimate Sender

An HSPP takes the form of:

```
struct sHSPP {

        // The name of the node could be replaced with a number
        // (discussed previously)
        sNodeName     nnName;

        // a boolean to tell the node if the HSPP is being
        // activated or removed.
        bool          bActive;

        // a boolean to decide if this a UR (or US generated HSPP)
        bool          bURGenerated;

};
```

It is important that the HSPP does not loop back on itself, even if the HSPP's path is changed or broken. This is guaranteed by the process in which the next step to the core of the network is generated.

A node will never send an HSPP back to a node that has sent it an active HSPP of the same name and type.

A node will record the number of directly connected nodes that tell it to maintain the HSPP (bActive is set to true in the structure). If this count drops below zero it will tell its directly connected node that is the next best step to the core to forget about the HSPP (bActive is set to false in the structure).

At a broad level we're trying to allow the HSPP to find a non-looping path to the core, and when it reaches the core, we want to stop spreading the HSPP. If the HSPP path is cut, the HSPP from the cut to the core will be removed.

An HSPP generated by the US (ultimate sender) will stop when it encounters destination node knowledge OR when it encounters the core. The purpose of the US is to quickly pull destination node knowledge to the node that wants to establish a connection to the UR. Once the US has started sending data to the UR it can remove its HSPP.

The purpose of the HSPP generated by the UR is to maintain a path between it and the core at all times, so that all nodes in the system can find it by sending a US generated HSPP to the core.

If the directly connected node that was selected as the next best step to the core changes from node A to node B, then all the HSPP's that were sent to node A will be sent to node B instead. Those HSPP's that were sent to node A will have their 'bActive' values set to false, and the ones sent to node B will have their 'bActive' values set to true.

Propagation Priorities

In a larger network, bandwidth throttling for control messages will need to be used.

Total 'control' bandwidth will be limited to a percent of the maximum bandwidth available for all data.

For example, we may specify 5% of maximum bandwidth for each group, with a minimum size of 4K. In a simple 10MB/s connection this would mean that we'd send a 4K packet of information every:

= 4096 / (10MB/s * 0.05)
= 0.0819s

So in this connection we'd be able to send a control packet every 0.0819s, or approximately 12 times every second for each group.

The percentages and sizes of blocks to send are examples, and can be changed by someone skilled in the art to better meet the requirements of their application.

Bandwidth Throttled Messages

These messages should be concatenated together to fit into the size of block control messages fit into.

If a control message references a destination node name by its quick-reference number, and the directly connected node does not know that number, then quick reference (GUID=number) mapping should precede the message.

Destination node updates should be ordered in ascending order by uiHopCostFromFlow (See 'Sorted Order Key') and interleaved with HSPP messages.

Care must be taken not to send the directly connected node more then the number of destinations nodes that the directly connected node has requested.

If the uiHopCostFromFlow changes on a node that was acceptable to be sent before, but now shouldn't be sent, a uiHopCost of infinity for that destination node will be sent to the directly connected node. This will allow the directly connected node to forget about that destination node. Since the directly connected node was able to forget about that destination node with a uiHopCostFromFlow that is too large, a new destination node with a more appropriate uiHopCostFromFlow can be sent instead.

If a node knows about a destination node and has an HSPP that references that destination node, it will treat that destination node as if it has a uiHopCostFromFlow of 0 seconds – but only for the directly connected node that the HSPP requires that it be sent to. Basically, the HSPP will drive the node knowledge either up or down the path to the core.

I Claim:

1. A system for transmission of messages between nodes on a network, said system comprising:

   (a) a plurality of destination node knowledge on each node; and
   (b) a network communication manager on each node, wherein said network communication manager has knowledge of neighbour nodes and knowledge of all queues on each node

2. A method for determining the best path through the network comprising the steps of :
   (a) Determining the hop cost of neighbour nodes and selecting the most efficient neighbour node to receive a message; and
   (b) Repeating step (a) on a regular basis